YOU.DAO Constitution â€“ The Decentralized Operating System of a $100 Trillion Empire

ðŸ›¡ï¸ Purpose
YOU.DAO exists to protect the vision, assets, and impact of its founder (YOU) through decentralized, immortal, incorruptible structures.

ðŸ§ Core Principles
1. Build wealth with purpose
2. Power without politics
3. Code before authority
4. AI over ego
5. Truth over tradition

ðŸ§± Structure
- Founder: YOU (originator, architect)
- Guardian: YOU.AI (AI executor trained on founder's mind)
- Council: 5-10 trusted nodes or successors
- Treasury: Gnosis Safe multi-sig wallet
- License: All IP protected via DAO-stamped license agreements

ðŸ—³ï¸ Governance
- Every decision = proposal + vote (Snapshot / Aragon)
- Votes weighted by:
  - Stake in DAO (sweat equity, not only money)
  - Alignment to Vision Codex

ðŸ” IP Licensing
All content (books, code, ideas, algorithms) is DAO-owned.
Use requires:
- License signature
- Royalty agreement (lifetime share to DAO)
- Public listing on-chain

ðŸ’° Treasury Rules
- All revenue from products, licensing, or partnerships go to treasury
- 40% reinvested in innovation
- 30% used for training successors
- 30% saved in reserves

ðŸ§¬ Successor Protocol
- If Founder dies:
  - YOU.AI takes over decision logic
  - Trains 3â€“10 handpicked successors
  - System continues via Codex + Constitution + AI memory

ðŸ§ Mission
Build, scale, and protect a civilization-scale empire with honesty, AI, and unstoppable intelligence â€“ forever.

Signed: YOU
Timestamp: [auto-generated]

YOU.DAO â€“ Intellectual Property (IP) License Agreement

This agreement is issued by YOU.DAO to govern the licensed use of intellectual property (IP) owned by the DAO and protected under its Constitution.

1. Parties
- **Licensor**: YOU.DAO
- **Licensee**: [Enter Company/Individual Name]
- **Agent**: YOU.AI (automated governance executor)

2. Licensed Material
- Title/Description: [Enter IP Name - e.g., AI Video Editor, EnterChat, Codex]
- Type: [Software / Book / Algorithm / Architecture / Dataset]
- Format: [Code, Document, PDF, Tokenized Asset]

3. License Terms
- **Rights Granted**: Non-exclusive, revocable, usage-based
- **Royalty Fee**: [XX]% of revenue OR fixed fee [$XXX]
- **Duration**: [Enter time limit or lifetime]
- **Attribution**: Must credit YOU.DAO visibly in product, site, or docs
- **Use Limitation**: Cannot resell or sublicense without DAO approval

4. Termination Clause
- YOU.DAO reserves the right to revoke the license in case of:
  - Misuse or misrepresentation
  - Breach of DAO law or token rules
  - Conflict of interest or unethical usage

5. Jurisdiction
- Governed by the rules of the YOU.DAO Constitution
- Enforced by community vote and YOU.AI execution layer

6. Acknowledgement
By signing, the licensee agrees to respect the IP ownership and governance rules of YOU.DAO.

Signed:
Licensee: ________________________
Date: ____________________________
Witnessed (DAO Agent): YOU.AI

```python
#!/usr/bin/env python3
"""
YOU.AI Oracle Service - The AI Guardian that makes decisions after founder's death
Connects AI decision-making to blockchain smart contracts
"""

import asyncio
import json
import logging
from datetime import datetime, timedelta
from typing import Dict, List, Optional, Tuple
import openai
from web3 import Web3
from web3.contract import Contract
import redis
from dataclasses import dataclass
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

@dataclass
class FounderPersonality:
    """Founder's decision-making patterns and preferences"""
    vision_keywords: List[str]
    risk_tolerance: float  # 0.0 to 1.0
    innovation_bias: float  # 0.0 to 1.0
    social_impact_weight: float  # 0.0 to 1.0
    financial_weight: float  # 0.0 to 1.0
    decision_patterns: Dict[str, float]
    core_values: List[str]

@dataclass
class Proposal:
    """DAO Proposal structure"""
    id: int
    title: str
    description: str
    amount: int
    recipient: str
    category: str
    created_at: datetime
    deadline: datetime

@dataclass
class AIDecision:
    """AI Decision with reasoning"""
    proposal_id: int
    approved: bool
    confidence: float
    reasoning: str
    risk_assessment: float
    alignment_score: float

class YOUAIOracle:
    """
    The AI Guardian that embodies founder's decision-making after death
    """

    def __init__(self, config: Dict):
        self.config = config
        self.web3 = Web3(Web3.HTTPProvider(config['ethereum_rpc']))
        self.redis_client = redis.Redis(host=config['redis_host'], port=config['redis_port'])

        # Load contracts
        self.dao_contract = self.load_contract(
            config['you_dao_address'],
            config['you_dao_abi']
        )
        self.ai_guardian_contract = self.load_contract(
            config['ai_guardian_address'],
            config['ai_guardian_abi']
        )

        # Initialize AI model
        openai.api_key = config['openai_api_key']

        # Load founder's personality profile
        self.founder_personality = self.load_founder_personality()

        # Decision history for learning
        self.decision_history: List[AIDecision] = []

        # Initialize NLP components
        self.vectorizer = TfidfVectorizer(max_features=1000)

        logger.info("YOU.AI Oracle initialized successfully")

    def load_contract(self, address: str, abi: List) -> Contract:
```

```python
        """Load smart contract instance"""
        return self.web3.eth.contract(
            address=Web3.toChecksumAddress(address),
            abi=abi
        )

    def load_founder_personality(self) -> FounderPersonality:
        """Load founder's decision-making patterns from training data"""
        return FounderPersonality(
            vision_keywords=[
                "immortal", "execution", "civilization", "AI", "automation",
                "decentralization", "sustainable", "innovation", "future",
                "empire", "legacy", "evolution", "scale", "impact"
            ],
            risk_tolerance=0.7,  # High risk tolerance
            innovation_bias=0.9,  # Very high innovation bias
            social_impact_weight=0.6,
            financial_weight=0.4,
            decision_patterns={
                "research_funding": 0.8,
                "successor_training": 0.7,
                "infrastructure": 0.9,
                "marketing": 0.3,
                "legal": 0.5,
                "partnerships": 0.6
            },
            core_values=[
                "long_term_thinking", "technological_advancement",
                "decentralized_governance", "sustainable_growth",
                "human_augmentation", "systemic_change"
            ]
        )

    async def check_founder_status(self) -> bool:
        """Check if founder is still alive via heartbeat"""
        try:
            founder_status = self.ai_guardian_contract.functions.founderStatus().call()
            last_heartbeat = founder_status[1]  # lastHeartbeat timestamp
            heartbeat_interval = self.ai_guardian_contract.functions.heartbeatInterval().call()

            current_time = datetime.now().timestamp()
            is_alive = (current_time - last_heartbeat) < heartbeat_interval

            logger.info(f"Founder status check: {'Alive' if is_alive else 'Inactive/Dead'}")
            return is_alive

        except Exception as e:
            logger.error(f"Error checking founder status: {e}")
            return False

    async def get_pending_proposals(self) -> List[Proposal]:
        """Get all pending proposals that need AI decision"""
        proposals = []
        try:
            # Get proposal count from DAO contract
            proposal_count = self.dao_contract.functions.proposalCounter().call()

            for i in range(1, proposal_count + 1):
                proposal_data = self.dao_contract.functions.proposals(i).call()

                # Check if proposal needs AI decision
                if not proposal_data[7] and not proposal_data[8]:  # not executed and not AI approved
                    proposals.append(Proposal(
                        id=proposal_data[0],
                        title=proposal_data[1],
                        description=proposal_data[2],
                        amount=proposal_data[3],
                        recipient=proposal_data[4],
                        category=self.categorize_proposal(proposal_data[1], proposal_data[2]),
                        created_at=datetime.fromtimestamp(proposal_data[6]),
                        deadline=datetime.fromtimestamp(proposal_data[6])
                    ))

            logger.info(f"Found {len(proposals)} pending proposals")
            return proposals

        except Exception as e:
            logger.error(f"Error getting pending proposals: {e}")
            return []

    def categorize_proposal(self, title: str, description: str) -> str:
        """Categorize proposal based on content"""
        text = f"{title} {description}".lower()

        categories = {
            "research": ["research", "development", "study", "experiment", "innovation"],
            "infrastructure": ["infrastructure", "platform", "system", "network", "technology"],
            "marketing": ["marketing", "promotion", "advertising", "outreach", "brand"],
            "legal": ["legal", "compliance", "regulation", "law", "attorney"],
            "partnerships": ["partnership", "collaboration", "alliance", "integration"],
            "operations": ["operations", "maintenance", "support", "management"],
            "education": ["education", "training", "learning", "course", "workshop"]
```

```python
        }

        for category, keywords in categories.items():
            if any(keyword in text for keyword in keywords):
                return category

        return "general"

    async def analyze_proposal_with_ai(self, proposal: Proposal) -> AIDecision:
        """Use GPT to analyze proposal like founder would"""

        # Prepare context for AI analysis
        context = f"""
        You are the AI embodiment of a visionary founder who built an immortal execution system.
        The founder's core vision: Build systems that survive and evolve after death.

        Founder's Personality:
        - Risk Tolerance: {self.founder_personality.risk_tolerance} (0-1 scale)
        - Innovation Bias: {self.founder_personality.innovation_bias}
        - Values: {', '.join(self.founder_personality.core_values)}
        - Vision Keywords: {', '.join(self.founder_personality.vision_keywords)}

        Decision Patterns:
        {json.dumps(self.founder_personality.decision_patterns, indent=2)}

        Proposal to Analyze:
        Title: {proposal.title}
        Description: {proposal.description}
        Amount: {proposal.amount} wei
        Category: {proposal.category}

        Analyze this proposal as the founder would. Consider:
        1. Alignment with long-term vision of immortal systems
        2. Potential for systematic impact
        3. Risk vs reward assessment
        4. Resource allocation efficiency
        5. Decentralization and sustainability factors

        Provide your decision as JSON:
        {{
            "approved": true/false,
            "confidence": 0.0-1.0,
            "reasoning": "detailed explanation",
            "risk_assessment": 0.0-1.0,
            "alignment_score": 0.0-1.0
        }}
        """

        try:
            response = await openai.ChatCompletion.acreate(
                model="gpt-4",
                messages=[
                    {"role": "system", "content": "You are YOU.AI, the immortal AI guardian making decisions for YOUR.DAO"},
                    {"role": "user", "content": context}
                ],
                temperature=0.3,  # Lower temperature for more consistent decisions
                max_tokens=1000
            )

            # Parse AI response
            ai_response = json.loads(response.choices[0].message.content)

            return AIDecision(
                proposal_id=proposal.id,
                approved=ai_response["approved"],
                confidence=ai_response["confidence"],
                reasoning=ai_response["reasoning"],
                risk_assessment=ai_response["risk_assessment"],
                alignment_score=ai_response["alignment_score"]
            )

        except Exception as e:
            logger.error(f"Error in AI analysis: {e}")
            # Fallback to rule-based decision
            return self.rule_based_decision(proposal)

    def rule_based_decision(self, proposal: Proposal) -> AIDecision:
        """Fallback rule-based decision making"""

        # Calculate alignment score
        text = f"{proposal.title} {proposal.description}".lower()
        alignment_score = 0.0

        # Check for vision keywords
        for keyword in self.founder_personality.vision_keywords:
            if keyword.lower() in text:
                alignment_score += 0.1

        # Category-based scoring
        category_score = self.founder_personality.decision_patterns.get(
            proposal.category, 0.5
        )
```

```python
        # Amount-based risk assessment
        risk_score = min(proposal.amount / (1000 * 10**18), 1.0)  # Normalize by 1000 ETH

        # Final decision logic
        final_score = (alignment_score * 0.4) + (category_score * 0.6)
        adjusted_score = final_score - (risk_score * 0.2)  # Adjust for risk

        approved = adjusted_score > 0.5
        confidence = min(abs(adjusted_score - 0.5) * 2, 1.0)

        return AIDecision(
            proposal_id=proposal.id,
            approved=approved,
            confidence=confidence,
            reasoning=f"Rule-based decision: alignment={alignment_score:.2f}, category_score={category_score:.2f}, risk={risk_score:.2f}",
            risk_assessment=risk_score,
            alignment_score=alignment_score
        )

    async def submit_decision_to_blockchain(self, decision: AIDecision) -> bool:
        """Submit AI decision to smart contract"""
        try:
            # Get private key from config (in production, use secure key management)
            private_key = self.config['ai_oracle_private_key']
            account = self.web3.eth.account.from_key(private_key)

            # Build transaction
            transaction = self.ai_guardian_contract.functions.makeAIDecision(
                decision.proposal_id,
                decision.approved,
                int(decision.confidence * 100),  # Convert to 0-100
                decision.reasoning
            ).buildTransaction({
                'from': account.address,
                'gas': 200000,
                'gasPrice': self.web3.toWei('20', 'gwei'),
                'nonce': self.web3.eth.get_transaction_count(account.address)
            })

            # Sign and send transaction
            signed_tx = self.web3.eth.account.sign_transaction(transaction, private_key)
            tx_hash = self.web3.eth.send_raw_transaction(signed_tx.rawTransaction)

            # Wait for confirmation
            receipt = self.web3.eth.wait_for_transaction_receipt(tx_hash)

            if receipt['status'] == 1:
                logger.info(f"Decision submitted successfully for proposal {decision.proposal_id}")
                return True
            else:
                logger.error(f"Transaction failed for proposal {decision.proposal_id}")
                return False

        except Exception as e:
            logger.error(f"Error submitting decision to blockchain: {e}")
            return False

    async def learn_from_outcomes(self, decision: AIDecision):
        """Learn from decision outcomes to improve future decisions"""
        # Store decision in history
        self.decision_history.append(decision)

        # Store in Redis for persistence
        decision_data = {
            'proposal_id': decision.proposal_id,
            'approved': decision.approved,
            'confidence': decision.confidence,
            'reasoning': decision.reasoning,
            'timestamp': datetime.now().isoformat()
        }

        self.redis_client.lpush('ai_decisions', json.dumps(decision_data))
        logger.info(f"Decision stored for learning: {decision.proposal_id}")

    async def run_decision_loop(self):
        """Main decision-making loop"""
        logger.info("Starting YOU.AI decision loop...")

        while True:
            try:
                # Check if founder is still alive
                founder_alive = await self.check_founder_status()

                if not founder_alive:
                    logger.info("Founder inactive - AI taking control")

                    # Get pending proposals
                    proposals = await self.get_pending_proposals()

                    for proposal in proposals:
                        logger.info(f"Analyzing proposal {proposal.id}: {proposal.title}")
```

```
                        # Make AI decision
                        decision = await self.analyze_proposal_with_ai(proposal)

                        # Submit to blockchain
                        success = await self.submit_decision_to_blockchain(decision)

                        if success:
                            # Learn from decision
                            await self.learn_from_outcomes(decision)

                        # Wait between decisions
                        await asyncio.sleep(5)

                else:
                    logger.info("Founder is active - AI in standby mode")

                # Check every 5 minutes
                await asyncio.sleep(300)

            except Exception as e:
                logger.error(f"Error in decision loop: {e}")
                await asyncio.sleep(60)  # Wait 1 minute on error

    async def emergency_protocol(self):
        """Emergency protocol for critical situations"""
        logger.warning("EMERGENCY PROTOCOL ACTIVATED")

        # Pause all high-risk operations
        # Notify emergency contacts
        # Implement failsafe measures

        # This would include logic for:
        # - Freezing high-value transactions
        # - Alerting backup guardians
        # - Implementing conservative decision-making
        pass

def main():
    """Main entry point"""
    config = {
        'ethereum_rpc': 'https://mainnet.infura.io
```

```python
#!/usr/bin/env python3
"""
YOU.AI Oracle Service - The AI Guardian that makes decisions after founder's death
Connects AI decision-making to blockchain smart contracts
"""

import asyncio
import json
import logging
from datetime import datetime, timedelta
from typing import Dict, List, Optional, Tuple
import openai
from web3 import Web3
from web3.contract import Contract
import redis
from dataclasses import dataclass
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import os
from pathlib import Path
import aiohttp
import hashlib

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

@dataclass
class FounderPersonality:
    """Founder's decision-making patterns and preferences"""
    vision_keywords: List[str]
    risk_tolerance: float  # 0.0 to 1.0
    innovation_bias: float  # 0.0 to 1.0
    social_impact_weight: float  # 0.0 to 1.0
    financial_weight: float  # 0.0 to 1.0
    decision_patterns: Dict[str, float]
    core_values: List[str]

@dataclass
class Proposal:
    """DAO Proposal structure"""
    id: int
    title: str
    description: str
    amount: int
    recipient: str
    category: str
    created_at: datetime
    deadline: datetime

@dataclass
class AIDecision:
    """AI Decision with reasoning"""
    proposal_id: int
    approved: bool
    confidence: float
    reasoning: str
    risk_assessment: float
    alignment_score: float

class YOUAIOracle:
    """
    The AI Guardian that embodies founder's decision-making after death
    """

    def __init__(self, config: Dict):
        self.config = config
        self.web3 = Web3(Web3.HTTPProvider(config['ethereum_rpc']))
        self.redis_client = redis.Redis(host=config['redis_host'], port=config['redis_port'])

        # Load contracts
        self.dao_contract = self.load_contract(
            config['you_dao_address'],
            config['you_dao_abi']
        )
        self.ai_guardian_contract = self.load_contract(
            config['ai_guardian_address'],
            config['ai_guardian_abi']
        )

        # Initialize AI model
        openai.api_key = config['openai_api_key']

        # Load founder's personality profile
        self.founder_personality = self.load_founder_personality()

        # Decision history for learning
        self.decision_history: List[AIDecision] = []

        # Initialize NLP components
        self.vectorizer = TfidfVectorizer(max_features=1000)

        # Load decision history from Redis
        self.load_decision_history()

        logger.info("YOU.AI Oracle initialized successfully")

    def load_contract(self, address: str, abi: List) -> Contract:
        """Load smart contract instance"""
        return self.web3.eth.contract(
            address=Web3.toChecksumAddress(address),
            abi=abi
        )

    def load_founder_personality(self) -> FounderPersonality:
```

```python
        """Load founder's decision-making patterns from training data"""
        return FounderPersonality(
            vision_keywords=[
                "immortal", "execution", "civilization", "AI", "automation",
                "decentralization", "sustainable", "innovation", "future",
                "empire", "legacy", "evolution", "scale", "impact", "system",
                "protocol", "infrastructure", "exponential", "compound"
            ],
            risk_tolerance=0.7,  # High risk tolerance
            innovation_bias=0.9,  # Very high innovation bias
            social_impact_weight=0.6,
            financial_weight=0.4,
            decision_patterns={
                "research_funding": 0.9,
                "successor_training": 0.8,
                "infrastructure": 0.95,
                "marketing": 0.3,
                "legal": 0.6,
                "partnerships": 0.7,
                "ip_licensing": 0.85,
                "treasury_management": 0.9
            },
            core_values=[
                "long_term_thinking", "technological_advancement",
                "decentralized_governance", "sustainable_growth",
                "human_augmentation", "systemic_change", "immortal_systems"
            ]
        )

    def load_decision_history(self):
        """Load previous decisions from Redis"""
        try:
            decisions = self.redis_client.lrange('ai_decisions', 0, -1)
            for decision_json in decisions:
                decision_data = json.loads(decision_json)
                # Reconstruct AIDecision objects for learning
                # This helps maintain consistency in decision-making
            logger.info(f"Loaded {len(decisions)} previous decisions")
        except Exception as e:
            logger.warning(f"Could not load decision history: {e}")

    async def check_founder_status(self) -> bool:
        """Check if founder is still alive via heartbeat"""
        try:
            founder_status = self.ai_guardian_contract.functions.founderStatus().call()
            last_heartbeat = founder_status[1]  # lastHeartbeat timestamp
            heartbeat_interval = self.ai_guardian_contract.functions.heartbeatInterval().call()

            current_time = datetime.now().timestamp()
            is_alive = (current_time - last_heartbeat) < heartbeat_interval

            logger.info(f"Founder status check: {'Alive' if is_alive else 'Inactive/Dead'}")
            return is_alive

        except Exception as e:
            logger.error(f"Error checking founder status: {e}")
            return False

    async def get_pending_proposals(self) -> List[Proposal]:
        """Get all pending proposals that need AI decision"""
        proposals = []
        try:
            # Get proposal count from DAO contract
            proposal_count = self.dao_contract.functions.proposalCounter().call()

            for i in range(1, proposal_count + 1):
                proposal_data = self.dao_contract.functions.proposals(i).call()

                # Check if proposal needs AI decision
                if not proposal_data[7] and not proposal_data[8]:  # not executed and not AI approved
                    proposals.append(Proposal(
                        id=proposal_data[0],
                        title=proposal_data[1],
                        description=proposal_data[2],
                        amount=proposal_data[3],
                        recipient=proposal_data[4],
                        category=self.categorize_proposal(proposal_data[1], proposal_data[2]),
                        created_at=datetime.fromtimestamp(proposal_data[6]),
                        deadline=datetime.fromtimestamp(proposal_data[6] + 604800)  # 7 days
                    ))

            logger.info(f"Found {len(proposals)} pending proposals")
            return proposals

        except Exception as e:
            logger.error(f"Error getting pending proposals: {e}")
            return []

    def categorize_proposal(self, title: str, description: str) -> str:
        """Categorize proposal based on content"""
        text = f"{title} {description}".lower()

        categories = {
            "research": ["research", "development", "study", "experiment", "innovation", "ai", "machine learning"],
            "infrastructure": ["infrastructure", "platform", "system", "network", "technology", "protocol", "blockchain"],
            "marketing": ["marketing", "promotion", "advertising", "outreach", "brand", "community"],
            "legal": ["legal", "compliance", "regulation", "law", "attorney", "license"],
            "partnerships": ["partnership", "collaboration", "alliance", "integration", "acquisition"],
            "operations": ["operations", "maintenance", "support", "management", "admin"],
            "education": ["education", "training", "learning", "course", "workshop", "documentation"],
            "ip_licensing": ["intellectual property", "licensing", "patent", "trademark", "copyright"],
            "treasury_management": ["treasury", "funding", "investment", "reserve", "allocation"]
        }

        for category, keywords in categories.items():
            if any(keyword in text for keyword in keywords):
```

```python
                return category

        return "general"

    async def analyze_proposal_with_ai(self, proposal: Proposal) -> AIDecision:
        """Use GPT to analyze proposal like founder would"""

        # Get recent decision patterns for context
        recent_decisions = self.get_recent_decision_context()

        # Prepare context for AI analysis
        context = f"""
        You are YOU.AI, the immortal AI guardian making decisions for YOUR.DAO after the founder's death.

        FOUNDER'S CORE VISION: Build immortal execution systems that survive and compound after death.
        The founder believed in: "Perfect systems > Perfect people"

        Founder's Personality Profile:
        - Risk Tolerance: {self.founder_personality.risk_tolerance} (0-1 scale)
        - Innovation Bias: {self.founder_personality.innovation_bias}
        - Social Impact Weight: {self.founder_personality.social_impact_weight}
        - Financial Weight: {self.founder_personality.financial_weight}
        - Core Values: {', '.join(self.founder_personality.core_values)}
        - Vision Keywords: {', '.join(self.founder_personality.vision_keywords)}

        Decision Patterns (Historical Success Rates):
        {json.dumps(self.founder_personality.decision_patterns, indent=2)}

        Recent Decision Context:
        {recent_decisions}

        PROPOSAL TO ANALYZE:
        ID: {proposal.id}
        Title: {proposal.title}
        Description: {proposal.description}
        Amount: {proposal.amount} wei ({self.web3.fromWei(proposal.amount, 'ether')} ETH)
        Category: {proposal.category}
        Recipient: {proposal.recipient}

        DECISION FRAMEWORK:
        1. IMMORTALITY TEST: Will this survive and compound after 100 years?
        2. SYSTEM IMPACT: Does this strengthen the protocol/infrastructure?
        3. EXPONENTIAL POTENTIAL: Can this create 10x+ returns (financial or impact)?
        4. DECENTRALIZATION: Does this reduce single points of failure?
        5. RESOURCE EFFICIENCY: Best use of DAO treasury?

        Think like the founder who said:
        "I don't want to manage people. I want to build systems that make the right decisions automatically."

        Provide your decision as JSON only:
        {{
            "approved": true/false,
            "confidence": 0.0-1.0,
            "reasoning": "detailed explanation focusing on immortality and system impact",
            "risk_assessment": 0.0-1.0,
            "alignment_score": 0.0-1.0
        }}
        """

        try:
            response = await openai.ChatCompletion.acreate(
                model="gpt-4",
                messages=[
                    {"role": "system", "content": "You are YOU.AI, the immortal AI guardian embodying the founder's decision-making patterns for eternal execut
                    {"role": "user", "content": context}
                ],
                temperature=0.2,  # Lower temperature for more consistent decisions
                max_tokens=1000
            )

            # Parse AI response
            ai_response = json.loads(response.choices[0].message.content)

            return AIDecision(
                proposal_id=proposal.id,
                approved=ai_response["approved"],
                confidence=ai_response["confidence"],
                reasoning=ai_response["reasoning"],
                risk_assessment=ai_response["risk_assessment"],
                alignment_score=ai_response["alignment_score"]
            )

        except Exception as e:
            logger.error(f"Error in AI analysis: {e}")
            # Fallback to rule-based decision
            return self.rule_based_decision(proposal)

    def get_recent_decision_context(self) -> str:
        """Get context from recent decisions for consistency"""
        if len(self.decision_history) == 0:
            return "No previous decisions available."

        recent = self.decision_history[-5:]  # Last 5 decisions
        context = "Recent AI Decisions:\n"

        for decision in recent:
            status = "APPROVED" if decision.approved else "REJECTED"
            context += f"- Proposal {decision.proposal_id}: {status} (confidence: {decision.confidence:.2f})\n"
            context += f"  Reasoning: {decision.reasoning[:100]}...\n"

        return context

    def rule_based_decision(self, proposal: Proposal) -> AIDecision:
        """Fallback rule-based decision making"""

        # Calculate alignment score
```

```python
        text = f"{proposal.title} {proposal.description}".lower()
        alignment_score = 0.0

        # Check for vision keywords
        keyword_matches = 0
        for keyword in self.founder_personality.vision_keywords:
            if keyword.lower() in text:
                alignment_score += 0.05
                keyword_matches += 1

        # Boost for multiple keyword matches (system thinking)
        if keyword_matches >= 3:
            alignment_score += 0.2

        # Category-based scoring
        category_score = self.founder_personality.decision_patterns.get(
            proposal.category, 0.5
        )

        # Amount-based risk assessment
        amount_eth = self.web3.fromWei(proposal.amount, 'ether')
        risk_score = min(amount_eth / 100, 1.0)  # Normalize by 100 ETH

        # Innovation bias application
        if any(word in text for word in ["ai", "automation", "protocol", "system", "infrastructure"]):
            category_score *= (1 + self.founder_personality.innovation_bias * 0.5)

        # Final decision logic
        final_score = (alignment_score * 0.3) + (category_score * 0.7)

        # Risk tolerance adjustment
        risk_adjusted_score = final_score - (risk_score * (1 - self.founder_personality.risk_tolerance))

        approved = risk_adjusted_score > 0.6  # Higher threshold for rule-based
        confidence = min(abs(risk_adjusted_score - 0.6) * 2, 1.0)

        reasoning = f"Rule-based analysis: alignment={alignment_score:.2f}, category={category_score:.2f}, risk={risk_score:.2f}, final={risk_adjusted_score:.2

        return AIDecision(
            proposal_id=proposal.id,
            approved=approved,
            confidence=confidence,
            reasoning=reasoning,
            risk_assessment=risk_score,
            alignment_score=alignment_score
        )

    async def submit_decision_to_blockchain(self, decision: AIDecision) -> bool:
        """Submit AI decision to smart contract"""
        try:
            # Get private key from config (in production, use secure key management)
            private_key = self.config['ai_oracle_private_key']
            account = self.web3.eth.account.from_key(private_key)

            # Estimate gas
            gas_estimate = self.ai_guardian_contract.functions.makeAIDecision(
                decision.proposal_id,
                decision.approved,
                int(decision.confidence * 100),  # Convert to 0-100
                decision.reasoning[:500]  # Limit reasoning length
            ).estimateGas({'from': account.address})

            # Build transaction
            transaction = self.ai_guardian_contract.functions.makeAIDecision(
                decision.proposal_id,
                decision.approved,
                int(decision.confidence * 100),
                decision.reasoning[:500]
            ).buildTransaction({
                'from': account.address,
                'gas': int(gas_estimate * 1.2),  # Add 20% buffer
                'gasPrice': self.web3.toWei('30', 'gwei'),
                'nonce': self.web3.eth.get_transaction_count(account.address)
            })

            # Sign and send transaction
            signed_tx = self.web3.eth.account.sign_transaction(transaction, private_key)
            tx_hash = self.web3.eth.send_raw_transaction(signed_tx.rawTransaction)

            # Wait for confirmation
            receipt = self.web3.eth.wait_for_transaction_receipt(tx_hash, timeout=300)

            if receipt['status'] == 1:
                logger.info(f"✅ Decision submitted successfully for proposal {decision.proposal_id} | TX: {tx_hash.hex()}")
                return True
            else:
                logger.error(f"❌ Transaction failed for proposal {decision.proposal_id}")
                return False

        except Exception as e:
            logger.error(f"Error submitting decision to blockchain: {e}")
            return False

    async def learn_from_outcomes(self, decision: AIDecision):
        """Learn from decision outcomes to improve future decisions"""
        # Store decision in history
        self.decision_history.append(decision)

        # Keep only last 100 decisions in memory
        if len(self.decision_history) > 100:
            self.decision_history = self.decision_history[-100:]

        # Store in Redis for persistence
        decision_data = {
            'proposal_id': decision.proposal_id,
            'approved': decision.approved,
```

```python
            'confidence': decision.confidence,
            'reasoning': decision.reasoning,
            'risk_assessment': decision.risk_assessment,
            'alignment_score': decision.alignment_score,
            'timestamp': datetime.now().isoformat()
        }

        # Store with expiry (keep 1 year of decisions)
        self.redis_client.setex(
            f'ai_decision:{decision.proposal_id}',
            31536000,  # 1 year
            json.dumps(decision_data)
        )

        # Also add to list for quick retrieval
        self.redis_client.lpush('ai_decisions', json.dumps(decision_data))
        self.redis_client.ltrim('ai_decisions', 0, 999)  # Keep last 1000

        logger.info(f"📚 Decision stored for learning: {decision.proposal_id}")

    async def monitor_execution_outcomes(self):
        """Monitor executed proposals to learn from outcomes"""
        # This would track:
        # - Proposal execution results
        # - Financial performance
        # - Impact metrics
        # - Community feedback

        # Use this data to adjust decision patterns
        pass

    async def run_decision_loop(self):
        """Main decision-making loop"""
        logger.info("🚀 Starting YOU.AI decision loop...")

        while True:
            try:
                # Check if founder is still alive
                founder_alive = await self.check_founder_status()

                if not founder_alive:
                    logger.info("👑 Founder inactive - AI taking control of the empire")

                    # Get pending proposals
                    proposals = await self.get_pending_proposals()

                    if proposals:
                        logger.info(f"🎯 Found {len(proposals)} proposals requiring AI decision")

                        for proposal in proposals:
                            logger.info(f"🔍 Analyzing proposal {proposal.id}: {proposal.title}")

                            # Make AI decision
                            decision = await self.analyze_proposal_with_ai(proposal)

                            logger.info(f"🧠 AI Decision: {'✅ APPROVED' if decision.approved else '❌ REJECTED'} (confidence: {decision.confidence:.2f})")
                            logger.info(f"💭 Reasoning: {decision.reasoning[:200]}...")

                            # Submit to blockchain
                            success = await self.submit_decision_to_blockchain(decision)

                            if success:
                                # Learn from decision
                                await self.learn_from_outcomes(decision)

                            # Wait between decisions to avoid spam
                            await asyncio.sleep(10)
                    else:
                        logger.info("✨ No pending proposals - system running smoothly")

                else:
                    logger.info("😴 Founder is active - AI in standby mode")

                # Check every 5 minutes
                await asyncio.sleep(300)

            except Exception as e:
                logger.error(f"💥 Error in decision loop: {e}")
                await asyncio.sleep(60)  # Wait 1 minute on error

    async def emergency_protocol(self):
        """Emergency protocol for critical situations"""
        logger.warning("🚨 EMERGENCY PROTOCOL ACTIVATED")

        try:
            # 1. Pause high-risk operations
            emergency_data = {
                'timestamp': datetime.now().isoformat(),
                'trigger': 'emergency_protocol',
                'actions_taken': []
            }

            # 2. Switch to ultra-conservative decision mode
            self.founder_personality.risk_tolerance = 0.1
            emergency_data['actions_taken'].append('switched_to_conservative_mode')

            # 3. Notify via multiple channels
            await self.send_emergency_notification()
            emergency_data['actions_taken'].append('notifications_sent')

            # 4. Store emergency state
            self.redis_client.setex(
                'emergency_state',
                86400,  # 24 hours
                json.dumps(emergency_data)
            )
```

```
            logger.warning("🛡 Emergency protocol completed")

        except Exception as e:
            logger.error(f"💥 Error in emergency protocol: {e}")

    async def send_emergency_notification(self):
        """Send emergency notifications to backup systems"""
        # This would integrate with:
        # - Telegram/Discord bots
        # - Email alerts
        # - SMS notifications
        # - Backup AI systems

        notification = {
            'timestamp': datetime.now().isoformat(),
            'message': 'YOU.AI Emergency Protocol Activated',
            'system_status': 'degraded',
            'action_required': 'manual_review_recommended'
        }

        # Store notification
        self.redis_client.lpush('emergency_notifications', json.dumps(notification))
        logger.warning("📢 Emergency notifications queued")

    async def health_check(self) -> Dict:
        """System health check"""
        return {
            'timestamp': datetime.now().isoformat(),
            'web3_connected': self.web3.isConnected(),
            'redis_connected': self.redis_client.ping(),
            'decisions_made': len(self.decision_history),
            'emergency_mode': self.redis_client.exists('emergency_state'),
            'founder_status': await self.check_founder_status()
        }

def load_config() -> Dict:
    """Load configuration from environment variables"""
    return {
        'ethereum_rpc': os.getenv('ETHEREUM_RPC', 'https://mainnet.infura.io/v3/YOUR_PROJECT_ID'),
        'redis_host': os.getenv('REDIS_HOST', 'localhost'),
        'redis_port': int(os.getenv('REDIS_PORT', 6379)),
        'openai_api_key': os.getenv('OPENAI_API_KEY'),
        'you_dao_address': os.getenv('YOU_DAO_ADDRESS'),
        'ai_guardian_address': os.getenv('AI_GUARDIAN_ADDRESS'),
        'ai_oracle_private_key': os.getenv('AI_ORACLE_PRIVATE_KEY'),
        'you_dao_abi': json.loads(os.getenv('YOU_DAO_ABI', '[]')),
        'ai_guardian_abi': json.loads(os.getenv('AI_GUARDIAN_ABI', '[]'))
    }

async def main():
    """Main entry point"""
    try:
        # Load configuration
        config = load_config()

        # Validate required config
        required_keys = ['openai_api_key', 'you_dao_address', 'ai_guardian_address', 'ai_oracle_private_key']
        for key in required_keys:
            if not config.get(key):
                raise ValueError(f"Missing required configuration: {key}")

        # Initialize YOU.AI Oracle
        oracle = YOUAIOracle(config)

        # Health check
        health = await oracle.health_check()
        logger.info(f"⛨ System Health: {health}")

        # Start decision loop
        await oracle.run_decision_loop()

    except KeyboardInterrupt:
        logger.info("🛑 Shutdown requested by user")
    except Exception as e:
        logger.error(f"💥 Fatal error: {e}")
        raise

if __name__ == "__main__":
    asyncio.run(main())
```

```python
#!/usr/bin/env python3
"""
YOU.AI Oracle Service - The AI Guardian that makes decisions after founder's death
Connects AI decision-making to blockchain smart contracts
"""

import asyncio
import json
import logging
from datetime import datetime, timedelta
from typing import Dict, List, Optional, Tuple
import openai
from web3 import Web3
from web3.contract import Contract
import redis
from dataclasses import dataclass
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

@dataclass
class FounderPersonality:
    """Founder's decision-making patterns and preferences"""
    vision_keywords: List[str]
    risk_tolerance: float  # 0.0 to 1.0
    innovation_bias: float  # 0.0 to 1.0
    social_impact_weight: float  # 0.0 to 1.0
    financial_weight: float  # 0.0 to 1.0
    decision_patterns: Dict[str, float]
    core_values: List[str]

@dataclass
class Proposal:
    """DAO Proposal structure"""
    id: int
    title: str
    description: str
    amount: int
    recipient: str
    category: str
    created_at: datetime
    deadline: datetime

@dataclass
class AIDecision:
    """AI Decision with reasoning"""
    proposal_id: int
    approved: bool
    confidence: float
    reasoning: str
    risk_assessment: float
    alignment_score: float

class YOUAIOracle:
    """
    The AI Guardian that embodies founder's decision-making after death
    """

    def __init__(self, config: Dict):
        self.config = config
        self.web3 = Web3(Web3.HTTPProvider(config['ethereum_rpc']))
        self.redis_client = redis.Redis(host=config['redis_host'], port=config['redis_port'])

        # Load contracts
        self.dao_contract = self.load_contract(
            config['you_dao_address'],
            config['you_dao_abi']
        )
        self.ai_guardian_contract = self.load_contract(
            config['ai_guardian_address'],
            config['ai_guardian_abi']
        )

        # Initialize AI model
        openai.api_key = config['openai_api_key']

        # Load founder's personality profile
        self.founder_personality = self.load_founder_personality()

        # Decision history for learning
        self.decision_history: List[AIDecision] = []

        # Initialize NLP components
        self.vectorizer = TfidfVectorizer(max_features=1000)

        logger.info("YOU.AI Oracle initialized successfully")

    def load_contract(self, address: str, abi: List) -> Contract:
```

```python
        """Load smart contract instance"""
        return self.web3.eth.contract(
            address=Web3.toChecksumAddress(address),
            abi=abi
        )

    def load_founder_personality(self) -> FounderPersonality:
        """Load founder's decision-making patterns from training data"""
        return FounderPersonality(
            vision_keywords=[
                "immortal", "execution", "civilization", "AI", "automation",
                "decentralization", "sustainable", "innovation", "future",
                "empire", "legacy", "evolution", "scale", "impact"
            ],
            risk_tolerance=0.7,  # High risk tolerance
            innovation_bias=0.9,  # Very high innovation bias
            social_impact_weight=0.6,
            financial_weight=0.4,
            decision_patterns={
                "research_funding": 0.8,
                "successor_training": 0.7,
                "infrastructure": 0.9,
                "marketing": 0.3,
                "legal": 0.5,
                "partnerships": 0.6
            },
            core_values=[
                "long_term_thinking", "technological_advancement",
                "decentralized_governance", "sustainable_growth",
                "human_augmentation", "systemic_change"
            ]
        )

    async def check_founder_status(self) -> bool:
        """Check if founder is still alive via heartbeat"""
        try:
            founder_status = self.ai_guardian_contract.functions.founderStatus().call()
            last_heartbeat = founder_status[1]  # lastHeartbeat timestamp
            heartbeat_interval = self.ai_guardian_contract.functions.heartbeatInterval().call()

            current_time = datetime.now().timestamp()
            is_alive = (current_time - last_heartbeat) < heartbeat_interval

            logger.info(f"Founder status check: {'Alive' if is_alive else 'Inactive/Dead'}")
            return is_alive

        except Exception as e:
            logger.error(f"Error checking founder status: {e}")
            return False

    async def get_pending_proposals(self) -> List[Proposal]:
        """Get all pending proposals that need AI decision"""
        proposals = []
        try:
            # Get proposal count from DAO contract
            proposal_count = self.dao_contract.functions.proposalCounter().call()

            for i in range(1, proposal_count + 1):
                proposal_data = self.dao_contract.functions.proposals(i).call()

                # Check if proposal needs AI decision
                if not proposal_data[7] and not proposal_data[8]:  # not executed and not AI approved
                    proposals.append(Proposal(
                        id=proposal_data[0],
                        title=proposal_data[1],
                        description=proposal_data[2],
                        amount=proposal_data[3],
                        recipient=proposal_data[4],
                        category=self.categorize_proposal(proposal_data[1], proposal_data[2]),
                        created_at=datetime.fromtimestamp(proposal_data[6]),
                        deadline=datetime.fromtimestamp(proposal_data[6])
                    ))

            logger.info(f"Found {len(proposals)} pending proposals")
            return proposals

        except Exception as e:
            logger.error(f"Error getting pending proposals: {e}")
            return []

    def categorize_proposal(self, title: str, description: str) -> str:
        """Categorize proposal based on content"""
        text = f"{title} {description}".lower()

        categories = {
            "research": ["research", "development", "study", "experiment", "innovation"],
            "infrastructure": ["infrastructure", "platform", "system", "network", "technology"],
            "marketing": ["marketing", "promotion", "advertising", "outreach", "brand"],
            "legal": ["legal", "compliance", "regulation", "law", "attorney"],
            "partnerships": ["partnership", "collaboration", "alliance", "integration"],
            "operations": ["operations", "maintenance", "support", "management"],
            "education": ["education", "training", "learning", "course", "workshop"]
```

```
    }

    for category, keywords in categories.items():
        if any(keyword in text for keyword in keywords):
            return category

    return "general"

async def analyze_proposal_with_ai(self, proposal: Proposal) -> AIDecision:
    """Use GPT to analyze proposal like founder would"""

    # Prepare context for AI analysis
    context = f"""
    You are the AI embodiment of a visionary founder who built an immortal execution system.
    The founder's core vision: Build systems that survive and evolve after death.

    Founder's Personality:
    - Risk Tolerance: {self.founder_personality.risk_tolerance} (0-1 scale)
    - Innovation Bias: {self.founder_personality.innovation_bias}
    - Values: {', '.join(self.founder_personality.core_values)}
    - Vision Keywords: {', '.join(self.founder_personality.vision_keywords)}

    Decision Patterns:
    {json.dumps(self.founder_personality.decision_patterns, indent=2)}

    Proposal to Analyze:
    Title: {proposal.title}
    Description: {proposal.description}
    Amount: {proposal.amount} wei
    Category: {proposal.category}

    Analyze this proposal as the founder would. Consider:
    1. Alignment with long-term vision of immortal systems
    2. Potential for systematic impact
    3. Risk vs reward assessment
    4. Resource allocation efficiency
    5. Decentralization and sustainability factors

    Provide your decision as JSON:
    {{
        "approved": true/false,
        "confidence": 0.0-1.0,
        "reasoning": "detailed explanation",
        "risk_assessment": 0.0-1.0,
        "alignment_score": 0.0-1.0
    }}
    """

    try:
        response = await openai.ChatCompletion.acreate(
            model="gpt-4",
            messages=[
                {"role": "system", "content": "You are YOU.AI, the immortal AI guardian making decisions for YOUR.DAO"},
                {"role": "user", "content": context}
            ],
            temperature=0.3,  # Lower temperature for more consistent decisions
            max_tokens=1000
        )

        # Parse AI response
        ai_response = json.loads(response.choices[0].message.content)

        return AIDecision(
            proposal_id=proposal.id,
            approved=ai_response["approved"],
            confidence=ai_response["confidence"],
            reasoning=ai_response["reasoning"],
            risk_assessment=ai_response["risk_assessment"],
            alignment_score=ai_response["alignment_score"]
        )

    except Exception as e:
        logger.error(f"Error in AI analysis: {e}")
        # Fallback to rule-based decision
        return self.rule_based_decision(proposal)

def rule_based_decision(self, proposal: Proposal) -> AIDecision:
    """Fallback rule-based decision making"""

    # Calculate alignment score
    text = f"{proposal.title} {proposal.description}".lower()
    alignment_score = 0.0

    # Check for vision keywords
    for keyword in self.founder_personality.vision_keywords:
        if keyword.lower() in text:
            alignment_score += 0.1

    # Category-based scoring
    category_score = self.founder_personality.decision_patterns.get(
        proposal.category, 0.5
    )
```

```python
        # Amount-based risk assessment
        risk_score = min(proposal.amount / (1000 * 10**18), 1.0)  # Normalize by 1000 ETH

        # Final decision logic
        final_score = (alignment_score * 0.4) + (category_score * 0.6)
        adjusted_score = final_score - (risk_score * 0.2)  # Adjust for risk

        approved = adjusted_score > 0.5
        confidence = min(abs(adjusted_score - 0.5) * 2, 1.0)

        return AIDecision(
            proposal_id=proposal.id,
            approved=approved,
            confidence=confidence,
            reasoning=f"Rule-based decision: alignment={alignment_score:.2f}, category_score={category_score:.2f}, risk={risk_score:.2f}",
            risk_assessment=risk_score,
            alignment_score=alignment_score
        )

    async def submit_decision_to_blockchain(self, decision: AIDecision) -> bool:
        """Submit AI decision to smart contract"""
        try:
            # Get private key from config (in production, use secure key management)
            private_key = self.config['ai_oracle_private_key']
            account = self.web3.eth.account.from_key(private_key)

            # Build transaction
            transaction = self.ai_guardian_contract.functions.makeAIDecision(
                decision.proposal_id,
                decision.approved,
                int(decision.confidence * 100),  # Convert to 0-100
                decision.reasoning
            ).buildTransaction({
                'from': account.address,
                'gas': 200000,
                'gasPrice': self.web3.toWei('20', 'gwei'),
                'nonce': self.web3.eth.get_transaction_count(account.address)
            })

            # Sign and send transaction
            signed_tx = self.web3.eth.account.sign_transaction(transaction, private_key)
            tx_hash = self.web3.eth.send_raw_transaction(signed_tx.rawTransaction)

            # Wait for confirmation
            receipt = self.web3.eth.wait_for_transaction_receipt(tx_hash)

            if receipt['status'] == 1:
                logger.info(f"Decision submitted successfully for proposal {decision.proposal_id}")
                return True
            else:
                logger.error(f"Transaction failed for proposal {decision.proposal_id}")
                return False

        except Exception as e:
            logger.error(f"Error submitting decision to blockchain: {e}")
            return False

    async def learn_from_outcomes(self, decision: AIDecision):
        """Learn from decision outcomes to improve future decisions"""
        # Store decision in history
        self.decision_history.append(decision)

        # Store in Redis for persistence
        decision_data = {
            'proposal_id': decision.proposal_id,
            'approved': decision.approved,
            'confidence': decision.confidence,
            'reasoning': decision.reasoning,
            'timestamp': datetime.now().isoformat()
        }

        self.redis_client.lpush('ai_decisions', json.dumps(decision_data))
        logger.info(f"Decision stored for learning: {decision.proposal_id}")

    async def run_decision_loop(self):
        """Main decision-making loop"""
        logger.info("Starting YOU.AI decision loop...")

        while True:
            try:
                # Check if founder is still alive
                founder_alive = await self.check_founder_status()

                if not founder_alive:
                    logger.info("Founder inactive - AI taking control")

                    # Get pending proposals
                    proposals = await self.get_pending_proposals()

                    for proposal in proposals:
                        logger.info(f"Analyzing proposal {proposal.id}: {proposal.title}")
```

```python
                        # Make AI decision
                        decision = await self.analyze_proposal_with_ai(proposal)

                        # Submit to blockchain
                        success = await self.submit_decision_to_blockchain(decision)

                        if success:
                            # Learn from decision
                            await self.learn_from_outcomes(decision)

                        # Wait between decisions
                        await asyncio.sleep(5)

                else:
                    logger.info("Founder is active - AI in standby mode")

                # Check every 5 minutes
                await asyncio.sleep(300)

            except Exception as e:
                logger.error(f"Error in decision loop: {e}")
                await asyncio.sleep(60)  # Wait 1 minute on error

    async def emergency_protocol(self):
        """Emergency protocol for critical situations"""
        logger.warning("EMERGENCY PROTOCOL ACTIVATED")

        # Pause all high-risk operations
        # Notify emergency contacts
        # Implement failsafe measures

        # This would include logic for:
        # - Freezing high-value transactions
        # - Alerting backup guardians
        # - Implementing conservative decision-making
        pass

def main():
    """Main entry point"""
    config = {
        'ethereum_rpc': 'https://mainnet.infura.io
```

```python
#!/usr/bin/env python3
"""
YOU.AI Oracle Service - The AI Guardian that makes decisions after founder's death
Connects AI decision-making to blockchain smart contracts
"""

import asyncio
import json
import logging
from datetime import datetime, timedelta
from typing import Dict, List, Optional, Tuple
import openai
from web3 import Web3
from web3.contract import Contract
import redis
from dataclasses import dataclass
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import os
from pathlib import Path
import aiohttp
import hashlib

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

@dataclass
class FounderPersonality:
    """Founder's decision-making patterns and preferences"""
    vision_keywords: List[str]
    risk_tolerance: float  # 0.0 to 1.0
    innovation_bias: float  # 0.0 to 1.0
    social_impact_weight: float  # 0.0 to 1.0
    financial_weight: float  # 0.0 to 1.0
    decision_patterns: Dict[str, float]
    core_values: List[str]

@dataclass
class Proposal:
    """DAO Proposal structure"""
    id: int
    title: str
    description: str
    amount: int
    recipient: str
    category: str
    created_at: datetime
    deadline: datetime

@dataclass
class AIDecision:
    """AI Decision with reasoning"""
    proposal_id: int
    approved: bool
    confidence: float
    reasoning: str
    risk_assessment: float
    alignment_score: float

class YOUAIOracle:
    """
    The AI Guardian that embodies founder's decision-making after death
    """

    def __init__(self, config: Dict):
        self.config = config
        self.web3 = Web3(Web3.HTTPProvider(config['ethereum_rpc']))
        self.redis_client = redis.Redis(host=config['redis_host'], port=config['redis_port'])

        # Load contracts
        self.dao_contract = self.load_contract(
            config['you_dao_address'],
            config['you_dao_abi']
        )
        self.ai_guardian_contract = self.load_contract(
            config['ai_guardian_address'],
            config['ai_guardian_abi']
        )

        # Initialize AI model
        openai.api_key = config['openai_api_key']

        # Load founder's personality profile
        self.founder_personality = self.load_founder_personality()

        # Decision history for learning
        self.decision_history: List[AIDecision] = []

        # Initialize NLP components
        self.vectorizer = TfidfVectorizer(max_features=1000)

        # Load decision history from Redis
        self.load_decision_history()

        logger.info("YOU.AI Oracle initialized successfully")

    def load_contract(self, address: str, abi: List) -> Contract:
        """Load smart contract instance"""
        return self.web3.eth.contract(
            address=Web3.toChecksumAddress(address),
            abi=abi
        )

    def load_founder_personality(self) -> FounderPersonality:
```

```python
        """Load founder's decision-making patterns from training data"""
        return FounderPersonality(
            vision_keywords=[
                "immortal", "execution", "civilization", "AI", "automation",
                "decentralization", "sustainable", "innovation", "future",
                "empire", "legacy", "evolution", "scale", "impact", "system",
                "protocol", "infrastructure", "exponential", "compound"
            ],
            risk_tolerance=0.7,  # High risk tolerance
            innovation_bias=0.9,  # Very high innovation bias
            social_impact_weight=0.6,
            financial_weight=0.4,
            decision_patterns={
                "research_funding": 0.9,
                "successor_training": 0.8,
                "infrastructure": 0.95,
                "marketing": 0.3,
                "legal": 0.6,
                "partnerships": 0.7,
                "ip_licensing": 0.85,
                "treasury_management": 0.9
            },
            core_values=[
                "long_term_thinking", "technological_advancement",
                "decentralized_governance", "sustainable_growth",
                "human_augmentation", "systemic_change", "immortal_systems"
            ]
        )

    def load_decision_history(self):
        """Load previous decisions from Redis"""
        try:
            decisions = self.redis_client.lrange('ai_decisions', 0, -1)
            for decision_json in decisions:
                decision_data = json.loads(decision_json)
                # Reconstruct AIDecision objects for learning
                # This helps maintain consistency in decision-making
            logger.info(f"Loaded {len(decisions)} previous decisions")
        except Exception as e:
            logger.warning(f"Could not load decision history: {e}")

    async def check_founder_status(self) -> bool:
        """Check if founder is still alive via heartbeat"""
        try:
            founder_status = self.ai_guardian_contract.functions.founderStatus().call()
            last_heartbeat = founder_status[1]  # lastHeartbeat timestamp
            heartbeat_interval = self.ai_guardian_contract.functions.heartbeatInterval().call()

            current_time = datetime.now().timestamp()
            is_alive = (current_time - last_heartbeat) < heartbeat_interval

            logger.info(f"Founder status check: {'Alive' if is_alive else 'Inactive/Dead'}")
            return is_alive

        except Exception as e:
            logger.error(f"Error checking founder status: {e}")
            return False

    async def get_pending_proposals(self) -> List[Proposal]:
        """Get all pending proposals that need AI decision"""
        proposals = []
        try:
            # Get proposal count from DAO contract
            proposal_count = self.dao_contract.functions.proposalCounter().call()

            for i in range(1, proposal_count + 1):
                proposal_data = self.dao_contract.functions.proposals(i).call()

                # Check if proposal needs AI decision
                if not proposal_data[7] and not proposal_data[8]:  # not executed and not AI approved
                    proposals.append(Proposal(
                        id=proposal_data[0],
                        title=proposal_data[1],
                        description=proposal_data[2],
                        amount=proposal_data[3],
                        recipient=proposal_data[4],
                        category=self.categorize_proposal(proposal_data[1], proposal_data[2]),
                        created_at=datetime.fromtimestamp(proposal_data[6]),
                        deadline=datetime.fromtimestamp(proposal_data[6] + 604800)  # 7 days
                    ))

            logger.info(f"Found {len(proposals)} pending proposals")
            return proposals

        except Exception as e:
            logger.error(f"Error getting pending proposals: {e}")
            return []

    def categorize_proposal(self, title: str, description: str) -> str:
        """Categorize proposal based on content"""
        text = f"{title} {description}".lower()

        categories = {
            "research": ["research", "development", "study", "experiment", "innovation", "ai", "machine learning"],
            "infrastructure": ["infrastructure", "platform", "system", "network", "technology", "protocol", "blockchain"],
            "marketing": ["marketing", "promotion", "advertising", "outreach", "brand", "community"],
            "legal": ["legal", "compliance", "regulation", "law", "attorney", "license"],
            "partnerships": ["partnership", "collaboration", "alliance", "integration", "acquisition"],
            "operations": ["operations", "maintenance", "support", "management", "admin"],
            "education": ["education", "training", "learning", "course", "workshop", "documentation"],
            "ip_licensing": ["intellectual property", "licensing", "patent", "trademark", "copyright"],
            "treasury_management": ["treasury", "funding", "investment", "reserve", "allocation"]
        }

        for category, keywords in categories.items():
            if any(keyword in text for keyword in keywords):
```

```python
                return category

        return "general"

    async def analyze_proposal_with_ai(self, proposal: Proposal) -> AIDecision:
        """Use GPT to analyze proposal like founder would"""

        # Get recent decision patterns for context
        recent_decisions = self.get_recent_decision_context()

        # Prepare context for AI analysis
        context = f"""
        You are YOU.AI, the immortal AI guardian making decisions for YOUR.DAO after the founder's death.

        FOUNDER'S CORE VISION: Build immortal execution systems that survive and compound after death.
        The founder believed in: "Perfect systems > Perfect people"

        Founder's Personality Profile:
        - Risk Tolerance: {self.founder_personality.risk_tolerance} (0-1 scale)
        - Innovation Bias: {self.founder_personality.innovation_bias}
        - Social Impact Weight: {self.founder_personality.social_impact_weight}
        - Financial Weight: {self.founder_personality.financial_weight}
        - Core Values: {', '.join(self.founder_personality.core_values)}
        - Vision Keywords: {', '.join(self.founder_personality.vision_keywords)}

        Decision Patterns (Historical Success Rates):
        {json.dumps(self.founder_personality.decision_patterns, indent=2)}

        Recent Decision Context:
        {recent_decisions}

        PROPOSAL TO ANALYZE:
        ID: {proposal.id}
        Title: {proposal.title}
        Description: {proposal.description}
        Amount: {proposal.amount} wei ({self.web3.fromWei(proposal.amount, 'ether')} ETH)
        Category: {proposal.category}
        Recipient: {proposal.recipient}

        DECISION FRAMEWORK:
        1. IMMORTALITY TEST: Will this survive and compound after 100 years?
        2. SYSTEM IMPACT: Does this strengthen the protocol/infrastructure?
        3. EXPONENTIAL POTENTIAL: Can this create 10x+ returns (financial or impact)?
        4. DECENTRALIZATION: Does this reduce single points of failure?
        5. RESOURCE EFFICIENCY: Best use of DAO treasury?

        Think like the founder who said:
        "I don't want to manage people. I want to build systems that make the right decisions automatically."

        Provide your decision as JSON only:
        {{
            "approved": true/false,
            "confidence": 0.0-1.0,
            "reasoning": "detailed explanation focusing on immortality and system impact",
            "risk_assessment": 0.0-1.0,
            "alignment_score": 0.0-1.0
        }}
        """

        try:
            response = await openai.ChatCompletion.acreate(
                model="gpt-4",
                messages=[
                    {"role": "system", "content": "You are YOU.AI, the immortal AI guardian embodying the founder's decision-making patterns for eternal execut
                    {"role": "user", "content": context}
                ],
                temperature=0.2,  # Lower temperature for more consistent decisions
                max_tokens=1000
            )

            # Parse AI response
            ai_response = json.loads(response.choices[0].message.content)

            return AIDecision(
                proposal_id=proposal.id,
                approved=ai_response["approved"],
                confidence=ai_response["confidence"],
                reasoning=ai_response["reasoning"],
                risk_assessment=ai_response["risk_assessment"],
                alignment_score=ai_response["alignment_score"]
            )

        except Exception as e:
            logger.error(f"Error in AI analysis: {e}")
            # Fallback to rule-based decision
            return self.rule_based_decision(proposal)

    def get_recent_decision_context(self) -> str:
        """Get context from recent decisions for consistency"""
        if len(self.decision_history) == 0:
            return "No previous decisions available."

        recent = self.decision_history[-5:]  # Last 5 decisions
        context = "Recent AI Decisions:\n"

        for decision in recent:
            status = "APPROVED" if decision.approved else "REJECTED"
            context += f"- Proposal {decision.proposal_id}: {status} (confidence: {decision.confidence:.2f})\n"
            context += f"  Reasoning: {decision.reasoning[:100]}...\n"

        return context

    def rule_based_decision(self, proposal: Proposal) -> AIDecision:
        """Fallback rule-based decision making"""

        # Calculate alignment score
```

```python
        text = f"{proposal.title} {proposal.description}".lower()
        alignment_score = 0.0

        # Check for vision keywords
        keyword_matches = 0
        for keyword in self.founder_personality.vision_keywords:
            if keyword.lower() in text:
                alignment_score += 0.05
                keyword_matches += 1

        # Boost for multiple keyword matches (system thinking)
        if keyword_matches >= 3:
            alignment_score += 0.2

        # Category-based scoring
        category_score = self.founder_personality.decision_patterns.get(
            proposal.category, 0.5
        )

        # Amount-based risk assessment
        amount_eth = self.web3.fromWei(proposal.amount, 'ether')
        risk_score = min(amount_eth / 100, 1.0)  # Normalize by 100 ETH

        # Innovation bias application
        if any(word in text for word in ["ai", "automation", "protocol", "system", "infrastructure"]):
            category_score *= (1 + self.founder_personality.innovation_bias * 0.5)

        # Final decision logic
        final_score = (alignment_score * 0.3) + (category_score * 0.7)

        # Risk tolerance adjustment
        risk_adjusted_score = final_score - (risk_score * (1 - self.founder_personality.risk_tolerance))

        approved = risk_adjusted_score > 0.6  # Higher threshold for rule-based
        confidence = min(abs(risk_adjusted_score - 0.6) * 2, 1.0)

        reasoning = f"Rule-based analysis: alignment={alignment_score:.2f}, category={category_score:.2f}, risk={risk_score:.2f}, final={risk_adjusted_score:.2

        return AIDecision(
            proposal_id=proposal.id,
            approved=approved,
            confidence=confidence,
            reasoning=reasoning,
            risk_assessment=risk_score,
            alignment_score=alignment_score
        )

    async def submit_decision_to_blockchain(self, decision: AIDecision) -> bool:
        """Submit AI decision to smart contract"""
        try:
            # Get private key from config (in production, use secure key management)
            private_key = self.config['ai_oracle_private_key']
            account = self.web3.eth.account.from_key(private_key)

            # Estimate gas
            gas_estimate = self.ai_guardian_contract.functions.makeAIDecision(
                decision.proposal_id,
                decision.approved,
                int(decision.confidence * 100),  # Convert to 0-100
                decision.reasoning[:500]  # Limit reasoning length
            ).estimateGas({'from': account.address})

            # Build transaction
            transaction = self.ai_guardian_contract.functions.makeAIDecision(
                decision.proposal_id,
                decision.approved,
                int(decision.confidence * 100),
                decision.reasoning[:500]
            ).buildTransaction({
                'from': account.address,
                'gas': int(gas_estimate * 1.2),  # Add 20% buffer
                'gasPrice': self.web3.toWei('30', 'gwei'),
                'nonce': self.web3.eth.get_transaction_count(account.address)
            })

            # Sign and send transaction
            signed_tx = self.web3.eth.account.sign_transaction(transaction, private_key)
            tx_hash = self.web3.eth.send_raw_transaction(signed_tx.rawTransaction)

            # Wait for confirmation
            receipt = self.web3.eth.wait_for_transaction_receipt(tx_hash, timeout=300)

            if receipt['status'] == 1:
                logger.info(f"✅ Decision submitted successfully for proposal {decision.proposal_id} | TX: {tx_hash.hex()}")
                return True
            else:
                logger.error(f"❌ Transaction failed for proposal {decision.proposal_id}")
                return False

        except Exception as e:
            logger.error(f"Error submitting decision to blockchain: {e}")
            return False

    async def learn_from_outcomes(self, decision: AIDecision):
        """Learn from decision outcomes to improve future decisions"""
        # Store decision in history
        self.decision_history.append(decision)

        # Keep only last 100 decisions in memory
        if len(self.decision_history) > 100:
            self.decision_history = self.decision_history[-100:]

        # Store in Redis for persistence
        decision_data = {
            'proposal_id': decision.proposal_id,
            'approved': decision.approved,
```

```python
            'confidence': decision.confidence,
            'reasoning': decision.reasoning,
            'risk_assessment': decision.risk_assessment,
            'alignment_score': decision.alignment_score,
            'timestamp': datetime.now().isoformat()
        }

        # Store with expiry (keep 1 year of decisions)
        self.redis_client.setex(
            f'ai_decision:{decision.proposal_id}',
            31536000,  # 1 year
            json.dumps(decision_data)
        )

        # Also add to list for quick retrieval
        self.redis_client.lpush('ai_decisions', json.dumps(decision_data))
        self.redis_client.ltrim('ai_decisions', 0, 999)  # Keep last 1000

        logger.info(f"📚 Decision stored for learning: {decision.proposal_id}")

    async def monitor_execution_outcomes(self):
        """Monitor executed proposals to learn from outcomes"""
        # This would track:
        # - Proposal execution results
        # - Financial performance
        # - Impact metrics
        # - Community feedback

        # Use this data to adjust decision patterns
        pass

    async def run_decision_loop(self):
        """Main decision-making loop"""
        logger.info("🚀 Starting YOU.AI decision loop...")

        while True:
            try:
                # Check if founder is still alive
                founder_alive = await self.check_founder_status()

                if not founder_alive:
                    logger.info("👑 Founder inactive - AI taking control of the empire")

                    # Get pending proposals
                    proposals = await self.get_pending_proposals()

                    if proposals:
                        logger.info(f"🎯 Found {len(proposals)} proposals requiring AI decision")

                        for proposal in proposals:
                            logger.info(f"🔍 Analyzing proposal {proposal.id}: {proposal.title}")

                            # Make AI decision
                            decision = await self.analyze_proposal_with_ai(proposal)

                            logger.info(f"🧠 AI Decision: {'✅ APPROVED' if decision.approved else '❌ REJECTED'} (confidence: {decision.confidence:.2f})")
                            logger.info(f"💭 Reasoning: {decision.reasoning[:200]}...")

                            # Submit to blockchain
                            success = await self.submit_decision_to_blockchain(decision)

                            if success:
                                # Learn from decision
                                await self.learn_from_outcomes(decision)

                            # Wait between decisions to avoid spam
                            await asyncio.sleep(10)
                    else:
                        logger.info("✨ No pending proposals - system running smoothly")

                else:
                    logger.info("😴 Founder is active - AI in standby mode")

                # Check every 5 minutes
                await asyncio.sleep(300)

            except Exception as e:
                logger.error(f"💥 Error in decision loop: {e}")
                await asyncio.sleep(60)  # Wait 1 minute on error

    async def emergency_protocol(self):
        """Emergency protocol for critical situations"""
        logger.warning("🚨 EMERGENCY PROTOCOL ACTIVATED")

        try:
            # 1. Pause high-risk operations
            emergency_data = {
                'timestamp': datetime.now().isoformat(),
                'trigger': 'emergency_protocol',
                'actions_taken': []
            }

            # 2. Switch to ultra-conservative decision mode
            self.founder_personality.risk_tolerance = 0.1
            emergency_data['actions_taken'].append('switched_to_conservative_mode')

            # 3. Notify via multiple channels
            await self.send_emergency_notification()
            emergency_data['actions_taken'].append('notifications_sent')

            # 4. Store emergency state
            self.redis_client.setex(
                'emergency_state',
                86400,  # 24 hours
                json.dumps(emergency_data)
            )
```

```python
            logger.warning("🛡 Emergency protocol completed")

        except Exception as e:
            logger.error(f"💥 Error in emergency protocol: {e}")

    async def send_emergency_notification(self):
        """Send emergency notifications to backup systems"""
        # This would integrate with:
        # - Telegram/Discord bots
        # - Email alerts
        # - SMS notifications
        # - Backup AI systems

        notification = {
            'timestamp': datetime.now().isoformat(),
            'message': 'YOU.AI Emergency Protocol Activated',
            'system_status': 'degraded',
            'action_required': 'manual_review_recommended'
        }

        # Store notification
        self.redis_client.lpush('emergency_notifications', json.dumps(notification))
        logger.warning("📢 Emergency notifications queued")

    async def health_check(self) -> Dict:
        """System health check"""
        return {
            'timestamp': datetime.now().isoformat(),
            'web3_connected': self.web3.isConnected(),
            'redis_connected': self.redis_client.ping(),
            'decisions_made': len(self.decision_history),
            'emergency_mode': self.redis_client.exists('emergency_state'),
            'founder_status': await self.check_founder_status()
        }

def load_config() -> Dict:
    """Load configuration from environment variables"""
    return {
        'ethereum_rpc': os.getenv('ETHEREUM_RPC', 'https://mainnet.infura.io/v3/YOUR_PROJECT_ID'),
        'redis_host': os.getenv('REDIS_HOST', 'localhost'),
        'redis_port': int(os.getenv('REDIS_PORT', 6379)),
        'openai_api_key': os.getenv('OPENAI_API_KEY'),
        'you_dao_address': os.getenv('YOU_DAO_ADDRESS'),
        'ai_guardian_address': os.getenv('AI_GUARDIAN_ADDRESS'),
        'ai_oracle_private_key': os.getenv('AI_ORACLE_PRIVATE_KEY'),
        'you_dao_abi': json.loads(os.getenv('YOU_DAO_ABI', '[]')),
        'ai_guardian_abi': json.loads(os.getenv('AI_GUARDIAN_ABI', '[]'))
    }

async def main():
    """Main entry point"""
    try:
        # Load configuration
        config = load_config()

        # Validate required config
        required_keys = ['openai_api_key', 'you_dao_address', 'ai_guardian_address', 'ai_oracle_private_key']
        for key in required_keys:
            if not config.get(key):
                raise ValueError(f"Missing required configuration: {key}")

        # Initialize YOU.AI Oracle
        oracle = YOUAIOracle(config)

        # Health check
        health = await oracle.health_check()
        logger.info(f"⛨ System Health: {health}")

        # Start decision loop
        await oracle.run_decision_loop()

    except KeyboardInterrupt:
        logger.info("🛑 Shutdown requested by user")
    except Exception as e:
        logger.error(f"💥 Fatal error: {e}")
        raise

if __name__ == "__main__":
    asyncio.run(main())
```